Exploring Game Balance in the Scandinavian Fox Game with Monte-Carlo Tree Search

Anton Janshagen
KTH
Sweden
anton.janshagen@gmail.com

Olof Mattsson
KTH
Sweden
olof.mattsson103@gmail.com

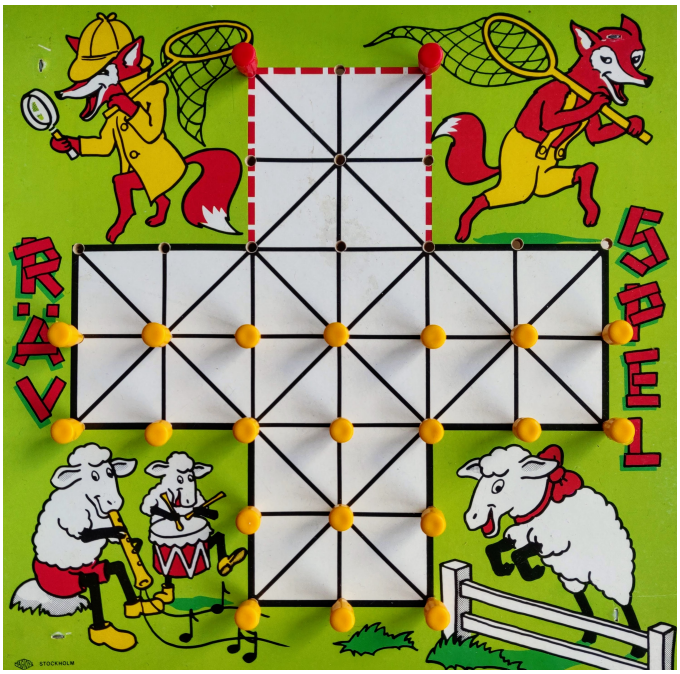


Figure 1: Modern adaptation from Magtoys (photo: Author)

ABSTRACT

This paper explores if Monte-Carlo Tree Search (MCTS) can perform well in Fox Game, a classic Scandinavian strategy game. MCTS is implemented using a cutoff in the simulation phase. The game state is then evaluated using a heuristic function that is formulated using theoretical arguments from its chess counterpart. MCTS is shown to perform on the same level as highly experienced human players, using limited computational resources. According to popular belief, as can be seen in online forums, the asymmetry in Fox Game leads to imbalances which favors the foxes. However the experiments in this paper show that, contrary to popular belief, it is the sheep that are favored, and quite heavily so. It is also shown that the game can be made more balanced by reducing the number of sheep at the start of the game from 20 to 18.

FDG '22, September 5–8, 2022, Athens, Greece

ACM ISBN 978-1-4503-9795-7/22/09…$15.00
https://doi.org/10.1145/3555858.3555919

CCS CONCEPTS

• **Computing methodologies** → **Game tree search**; **Heuristic function construction**; *Intelligent agents.*

KEYWORDS

Fox Games, Game Balance, Artificial Intelligence, Monte-Carlo Tree Search

ACM Reference Format:
Anton Janshagen and Olof Mattsson. 2022. Exploring Game Balance in the Scandinavian Fox Game with Monte-Carlo Tree Search. In *FDG '22: Proceedings of the 17th International Conference on the Foundations of Digital Games (FDG '22), September 5–8, 2022, Athens, Greece.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3555858.3555919

1 INTRODUCTION

The Fox Game (Swedish: Rävspel or Svälta räv) studied in this paper is a classic Scandinavian board game, and is said to have roots back to the 15th century, but it is unclear how similar those versions

were to the modern ones. Fox Game is a perfect information board game that, despite its simplicity, contains sophisticated strategy. It is similar to Chinese checkers and Fox-and-Geese in regards to both rules and game board.

The interest in Artificial Intelligence (AI) for board games has grown considerably in recent years. Monte-Carlo Tree Search (MCTS) specifically, is a search algorithm that has gained much popularity when making intelligent bots for board games. This is a form of AI that decides on the next move by a look-ahead search, scouting sequences of possible moves and counter-moves through a process of trial-and-error. In 2016 Deepminds AI AlphaGo, which combines MCTS and neural networks, was the first AI to beat a professional Go-player, which had been a longstanding goal in AI development [1]. Since then, similar combinations of MCTS and neural networks have achieved super-human strength in other demanding board games such as Hex, Shogi and chess. This paper examines if, and if so how, MCTS can be used to create a strong AI for Fox Game, and with said AI explore game balance and strategies for Fox Game.

To the best of our knowledge, there is no AI for the Scandinavian Fox Game in the literature. However, Fox-and-Geese, a related Fox Game, has been solved through a game specific algebraic decomposition, using exhaustive search on massive computers [2]. By contrast, in this paper, the Scandinavian Fox Game is analyzed using a more scalable and generic approach.

The main differences between Fox-and-Geese and the Scandinavian Fox Game are; (i) the former has only one fox, making the number of game states much smaller; (ii) in Fox-and-Geese the fox can only capture one goose per turn by jumping over it, while the fox in Fox Game can capture several sheep per turn; and (iii), the fox is never removed from the game in Fox-and-Geese, but in Fox Game one or both foxes can be captured. The complete set of rules for the Scandinavian Fox Game is described in section 2. All these features makes the Scandinavian Fox Game much more complex, and the method used in [2] infeasible.

The interest for Fox Game lies partly in its cultural value as a traditional Scandinavian board game, and partly in the fact that it is asymmetrical. Although several papers have been published about exploring board games with MCTS, [1] for example, they are mostly focused on symmetrical games such as Hex, Shogi or Chess. In this paper it is explored if MCTS is able to handle the effects of an asymmetric game, such as the fact that the players interact with the board in different ways, and that the number of moves available differs considerably.

The rest of the paper is organized as follows. Section 2 explains the history and rules of Fox Game. Section 3 describes how MCTS works, and explains certain variations of MCTS. Section 4 describes the heuristic function created in this paper to evaluate any board state in Fox Game. Section 5 describes how, and why, the experiments in the paper were made, as well as the obtained results of the experiments. In section 6, we discuss the results and other aspects of the paper. Finally, in section 7, we summarize the results of the paper.

2 FOX GAME

Fox Game is an old Scandinavian game that is a mixture of many different games. It has also evolved into different variants over the years. One effect of this is that there are several different sets of rules for the game, and since the game is not played competitively by professionals, there is no consensus on which are the best. In figure 1 a modern adaptation of Fox Game can be seen, and figure 2 shows the game board used in this paper in its starting position.

The following rules are the same in all variants of Fox Game, and variations of them will be discussed below. The game is a turn-based two player game, where one player plays as 20 sheep (purple pieces), and the other player plays as 2 foxes (red pieces). The goal of the sheep is to move 9 sheep into the pasture (marked in green) on the other side of the board. The foxes do not have a goal of their own, but they simply try to stop the sheep from achieving their goal. The sheep may only move one tile at a time, either side to side, or forward, but not diagonally or backwards. The foxes however may move one tile at a time along all lines on the board.

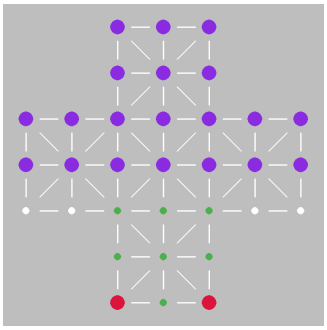
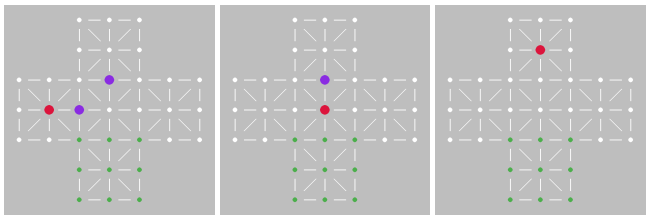


Figure 2: Our implementation of Fox Game in its starting position

The foxes can also capture sheep, and thereby remove them from the board. This can be done when a fox is standing next to a sheep, and the space on the other side of the sheep is empty. The fox may then move to the other side of the sheep while simultaneously removing the sheep from the board. If the fox is then put in a position where it may capture another sheep, it can immediately move and capture it without having to wait for the sheep to make a move first. See figure 3 for reference. The foxes cannot jump over each other.



(a) Starting position (b) Intermediate position (c) End position

Figure 3: Fox capturing two sheep in one turn

As mentioned above, there are certain alterations of the rules that deal with special situations that may arise when playing Fox Game. The following two rules can be found in many rule-books

online, but are not present in all of them. However we have included them in this paper for reasons described below.

The first one is that the foxes not only may jump over the sheep to capture them, but must do so if presented with the opportunity, even if the fox has already jumped once this turn. The purpose of this rule is to give the sheep the possibility to sacrifice a sheep in order to lure a fox from, or into, a certain position. Particularly this stops the foxes from staying in the pasture and physically stopping the sheep from entering, as the sheep can force the fox to move away. One instance of this scenario can be seen in figure 4.

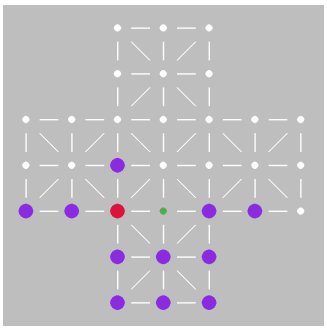


Figure 4: The fox is forced to jump out of the pasture which allows the sheep to win

Another additional rule described in [5] is that not only the sheep can be captured, but also the foxes. The foxes are not captured in the same way as the sheep. They are instead captured if, on their turn, they have no possible move. So if a fox has no available move on the start of its turn, it is removed from the board. This rule is necessary as the foxes otherwise can stay in the pasture to block the sheep from entering and thereby never lose. A fox being captured can be seen in figure 5.

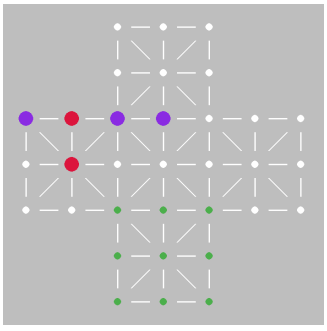


Figure 5: The upper fox cannot move and is thereby captured

A situation that is implicitly clear when humans play, but needs to be specified when bots play, is how to interpret the situation where the sheep cannot reach the pasture, but the foxes are not able to capture the sheep either. If for example the situation in figure 6 arises, the sheep has no reasonable chance to win, but they can stay back forever, and thereby never lose. Our solution to this was to set a limit of how many turns in a row the sheep may move sideways. If the sheep have not moved forward after a set number of turns they lose. We think this is a reasonable interpretation of the rules, as the foxes have successfully stopped the sheep from reaching the pasture. Throughout the paper the number of turns needed for the sheep to lose in this way is 10.

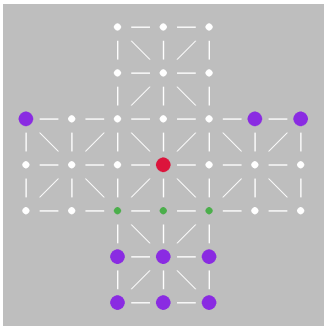


Figure 6: The sheep can prolong the game inevitably

3 MONTE-CARLO TREE SEARCH

MCTS is a best-first tree search algorithm [6]. The algorithm creates a search tree where it evaluates different nodes in the tree based on Monte-Carlo simulations, and gradually moves further down the tree to get more and more precise evaluations of the different states.

The first node of the tree is the root node. It represents the current game state, and is the only node without a parent node. The children of each node represent all of the legal moves that can be made from the board state corresponding to the parent node. Beside what the board state looks like, all nodes contain information about their evaluation, the number of times they have been visited, which player is to make the next move, and pointers to their parent and children nodes.

MCTS consists of four main steps which can be seen in figure 7. The first step is to select from which node, i.e. which game state, to visit next, this phase is called tree traversal. The process is done recursively from the root node until a node without children is reached. The selection between a node's children is done by picking the child with the highest UCB1-value according to equation (1), where Q_i is the value of the node, v_i is the number of times the node has been visited, and v_{pi} is the number of visits of the parent node [6]. The first term is the average value of the node per visit, and is called the exploitation term, and the second term is responsible for widening the search tree, and is called the exploration term. C is a constant that is chosen to balance the depth and width of the search tree. The optimal value of C will depend on the branching factor of the search tree, and is therefore game dependent. It can be seen from the UCB1 formula that a node with zero visits will have infinite UCB1-value, and will therefore always be chosen. If several children have infinite UCB1-value, one will be picked randomly.

$$UCB1_i = \frac{Q_i}{v_i} + C\sqrt{\frac{ln(v_{pi})}{v_i}} \quad (1)$$

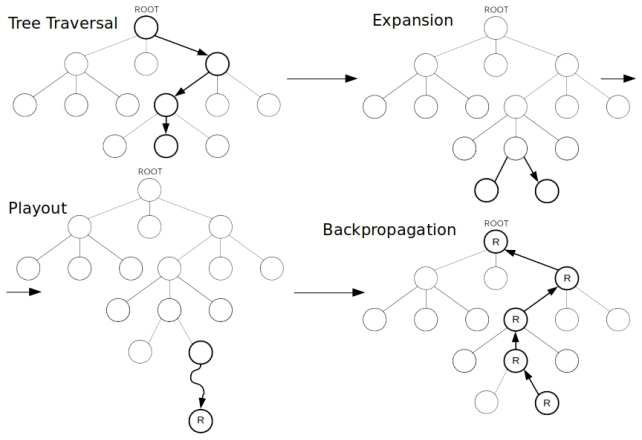


Figure 7: Overview of the Monte-Carlo Tree Search algorithm

When a leaf node has been reached in the tree traversal phase, the expansion phase starts. In this phase all of the possible child nodes are created, and a node is chosen at random.

The third phase is called playout. Here random moves are made from the game state associated with the leaf node, until the game ends. The result of the game is one for a win, and minus one for a loss.

The last part of the algorithm is backpropagation. The result from the playout is propagated back through the tree, and each visited node's value and number of visits is updated. Since every node is associated with a player, the value of a node must be updated based on if that player won the playout or not.

The process above is then repeated a certain number of times, known as the number of simulations. The more simulations you allow the algorithm, the more precise the evaluation will be. MCTS will not solve a game completely, but it will converge to the best solution given enough simulations [6]. The process can also be run for a specific amount of time, and the number of simulations will then vary depending on how long time one simulation takes.

After a desired number of simulations has been made, or after a certain amount of time, the most promising child node of the root is chosen as the move to make. This can be done in two different ways, either by picking the child of the root with highest average value per visit, or by picking the child with the most number of visits. Since the child with the highest average value will have many visits, according to equation (1), these approaches yield the same choice in most cases. In this paper the latter approach was used.

3.1 Cutoff

The idea of MCTS is that the nodes' values can be assessed with random simulations since a better position will lead to wins more often than a worse position when making random playouts from both positions. However this difference becomes smaller the more moves are made in the playout phase, because the difference drowns in the large randomness of the long playout. It can therefore be beneficial to introduce a cutoff in the playout phase after a certain number of random moves have been made. The game state can then be evaluated using a heuristic function. This can increase the performance of MCTS in games that require a large number of moves before either side wins, but it introduces the need for a heuristic function.

3.2 Heuristic Function

A heuristic function for a game can be made in different ways, and include many different metrics. The output of the heuristic function should represent the probability of victory based on the current game state, and is therefore the same for both players.

One common and traditional way to design a heuristic function is to consider both the value of the pieces left for each player, and where those pieces are situated on the board. The value of the remaining pieces is called the material part, and the value of the pieces' positions is called the positional part. An example of this can be read about in [3], where a heuristic function is made for chess. The evaluation compares, for example, how many pawns are left for each player, while also considering that defended pawns are worth more than those who are easily captured by the opponent.

A common dilemma with heuristic functions discussed in [6] is that more complex functions may give a more accurate evaluation of the board state, but they can as a trade off be more computationally expensive. This means that fewer simulations can be made in the same amount of time, and they may therefore produce worse results than simpler functions even if the evaluation from the complex heuristic function is better.

4 METHOD

All code was written in *Python 3.8* using the libraries *numpy* and *pygame* for calculations and visualization respectively.

4.1 Heuristic Function for Fox Game

The heuristic function used in this paper took inspiration from its chess counterpart in [3]. The material part consists of a weighted sum of the difference between the number of pieces of each sort for each player: $H_{mat} = w_f(F_1 - F_2) + w_s(S_1 - S_2)$. But since the players have zero pieces of one type the function simplifies to $H_{mat} = w_f F - w_s S$, where F is the number of foxes and S is the number of sheep. But we are really only interested in the relative value of the pieces, i.e. $\frac{w_f}{w_s}$. So one simplification that can be made is to set $w_s = 1$ and $w_f = q$, where q is how many sheep one fox is worth. This however means that the magnitude of the material part depends heavily on q, but the output should be a probability and thus be less than or equal to one. This was fixed by normalizing the coefficients with $(w_s + w_f)$, i.e. $(1 + q)$.

The positional part of our heuristic function used a piece-square table (PST) that assigns a value to each square. For every square that is occupied by a sheep, the associated value according to matrix (2) is summed to a variable V, which is used in the heuristic function in equation (3). The positional function only considers the sheep because observations showed that they needed help to know in which direction to move. Also a reasonable PST for the foxes is less obvious, and we wanted to influence the strategies of the AI as little as possible. The values of the PST were chosen by us based on prior knowledge of the game, but the idea is to encourage the sheep to move towards their goal.

$$PST = \begin{bmatrix} - & - & 0 & 0 & 0 & - & - \\ - & - & 1 & 1 & 1 & - & - \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 5 & 5 & 5 & 4 & 4 \\ - & - & 6 & 6 & 6 & - & - \\ - & - & 7 & 7 & 7 & - & - \end{bmatrix} \quad (2)$$

The positional part was reduced by a factor 0.1 so that the material and positional part have the same order of magnitude. The heuristic function also contains a parameter k, between zero and one, that balances the material and positional parts relative to each other.

All the variables in the heuristic function (F, S and V) were subtracted by their initial value to ensure that the function evaluates the game state to zero at the start of a game. The sum is also multiplied by a factor b that varies the overall magnitude of the function.

The heuristic function is mapped to values between negative one and one with the hyperbolic tangent function ($tanh$) to ensure that the result from a win or loss is worth more than a result from an evaluation by the heuristic function.

The heuristic function (3), is stated from the perspective of the foxes, so it is close to one if the foxes are winning, and close to negative one if the sheep are winning. The evaluation for the sheep is the negative value of equation (3).

$$\begin{aligned} H &= tanh(b[kH_{mat} + (1-k)H_{pst}]) \\ H_{mat} &= \frac{q}{1+q}(F-2) - \frac{1}{1+q}(S-20) \\ H_{pst} &= -0.1(V-38) \end{aligned} \quad (3)$$

5 EXPERIMENTS AND RESULTS

As a game lasts about 200 to 250 moves (when AIs play) a basic version of MCTS, without cutoff, would not be reasonable to use in the experiments. All experiments therefore used cutoff and a heuristic function. Most of the experiments were performed by letting agents with different constant-values play against each other. The agents played the same number of games with both pieces because, as can be seen in the results, the sheep are heavily favored.

The experiments considered how altering constant-values in the heuristic function changed the strength of the agents, and how the number of simulations and cutoff-value affected their performance. The AI was then tested against human players, and an attempt to balance the game was made.

The results that follow consists of approximately 1000 AI-played games, where some of them were played against humans.

5.1 Heuristic Function

The first experiment was to see how altering the different values (k, q, and b) in the heuristic function (3), and C in the UCB1 formula (1), affected the strength of an agent. A base agent with constant-values based on prior knowledge of the game and of MCTS was formulated, and agents with slightly different values played against it. The base agent used constant-values according to (4). Every altered agent played against the base agent 60 times, 30 times a sheep and 30 times as foxes.

Table 1: Results from agents with different constant-values playing against the base agent

Alteration	C = 0.5	C = 0.3	q = 14	q = 10
Base agent foxes wins	6	6	4	3
Base agent sheep wins	24	23	27	27
New agent foxes wins	5	7	3	3
New agent sheep wins	25	24	26	27
Base agent win rate	**0.50**	**0.48**	**0.50**	**0.53**

Alteration	k = 0.9	k = 0.7	b = 1.2	b = 0.8
Base agent foxes wins	8	11	3	2
Base agent sheep wins	24	26	25	25
New agent foxes wins	6	4	4	5
New agent sheep wins	22	19	28	28
Base agent win rate	**0.61**	**0.50**	**0.47**	**0.45**

$$\begin{aligned} k &= 0.8 \\ q &= 12 \\ b &= 1 \\ C &= 0.4 \end{aligned} \quad (4)$$

When the base agent ($k = 0.8$, $q = 12$, $b = 1$ and $C = 0.4$) played against agents with slight variations that affected the heuristic function the win rates in table 1 were obtained.

5.2 Simulations

For the next experiment we tested how the number of simulations affected the performance of the agent. This was done partly to demonstrate that the algorithm works as intended, as a greater number of simulations should increase the performance, and partly to find how its performance depends on the number of simulations. This was done in order to suggest a number of simulations that balances performance and time consumption.

The experiment was done by having a base agent, according to equation (4), with 1000 simulations play at least 30 matches against agents with a different number of simulations for each set of 30 matches, and recording the results. All agents used the same heuristic function as the base agent.

The win rate of agents with different numbers of simulations when playing against an opponent with 1000 simulations can be seen in figure 8. The win rate increases rapidly for agents with a low number of simulations, and for agents with a larger number of simulations the payoff in performance diminishes, this was an expected result.

5.3 Cutoff

The benefits of the playout phase is that the added randomness helps the algorithm find strategies that at first might seem inferior. But without a cutoff, or too high cutoff-value, too much randomness will make the algorithm worse. The playout is also the most computationally demanding phase of the algorithm. So the cutoff-value

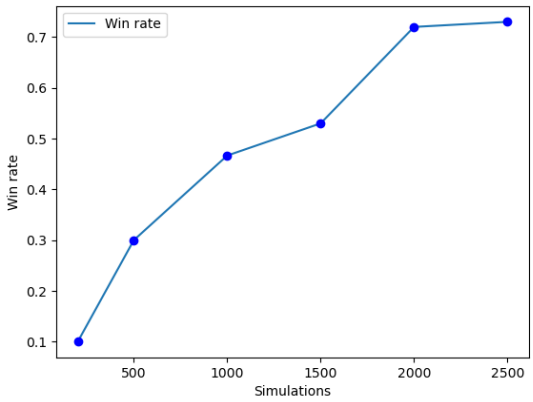


Figure 8: Win rate against agent with 1000 simulations for agents with different number of simulations

has to balance both the amount of randomness in the playout phase, as well as how much time is spent in the playout phase versus the other phases. It is possible that cutoff-value zero, i.e. immediate evaluation, is optimal because of the computations needed in the playout.

The cutoff experiments were divided into two parts. The first part used the base agent with a constant number of simulations (3000), but with different cutoff-values between different sets of matches. More than 30 matches were played between different agents, until an optimal cutoff-value was obtained.

This first part considered only how much randomness to introduce into the algorithm, while the second part focused on the time aspect of the cutoff. Here, the agents had the same amount of time to reach a decision, so the number of simulations had to vary accordingly. In the first experiment, with cutoff-value zero and five, the agents had 1.5 seconds, which amounted to about 4500 and 1100 simulations respectively. And in the second experiment, with cutoff-value zero and two, they had 1 second, which allowed for about 3000 and 1200 simulations respectively. In this second part of the experiment at least 50 matches were played between each pair of agents.

The goal of the second part of this experiment was still to find an optimal cutoff-value, but only cutoff-values less than the value found in part one was considered. This is because if a higher cutoff-value is worse than a lower one when the same number of simulations is used, it will definitely be worse when the same amount of time is given. However this is not necessarily true for a cutoff-value less than the optimal value found in part one because a lower cutoff-value allows for more simulations, which might be more important than having the optimal cutoff-value, if they are given the same amount of time.

In table 2 the win rates from part one and part two of the cutoff experiment can be found. In part one both agents used 3000 simulations, and in part two both agents had 1 or 1.5 seconds each to choose a move.

Table 2: Win rates of agents with different cutoffs

Limiting resource	3000 Simulations		
Cutoff: Agent 1 - Agent 2	0 - 5	5 - 10	10 - 20
Agent 1 foxes wins	0	3	0
Agent 1 sheep wins	11	13	15
Agent 2 foxes wins	8	2	0
Agent 2 sheep wins	19	12	15
Win rate agent 1	**0.29**	**0.53**	**0.50**

Limiting resource	1.5 Seconds	1 Second
Cutoff: Agent 1 - Agent 2	0-5	0 - 2
Agent 1 foxes wins	6	8
Agent 1 sheep wins	23	22
Agent 2 foxes wins	2	8
Agent 2 sheep wins	19	22
Win rate agent 1	**0.58**	**0.5**

5.4 Strength Versus Humans

To truly see the strength of the agent we let it play against humans. The agent that played against humans was the base agent from equation (4) with the optimal cutoff-value from section 5.3. The agent used 10000 simulations which gave it roughly three seconds to find a move. All human participants had prior experience of Fox Game. But since the game is not played competitively by professionals, we never found anyone with more experience than us to face the AI.

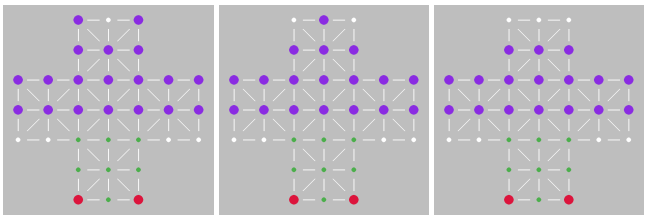
All human players with limited experience of Fox Game lost every game, regardless of whether they played as sheep or as foxes. We, as more experienced players, also lost every game when we played as foxes, but as sheep we have managed to win several games. At the moment of writing we estimate that we can win every game when playing as sheep, but none when playing as foxes.

5.5 Imbalance of Fox Game

As Fox Game is an asymmetrical game it is not certain that the foxes and the sheep have equal chance of winning. From the data gathered in the heuristic function experiment 5.1 and cutoff experiment 5.3, that both use high quality agents with many simulations, it can be seen that the sheep win 85% of games. So with the rules currently implementation makes Fox Game a very unbalanced game. It is therefore interesting to examine how the game can become more balanced. As all of the variations of the rules described in section 2 plays an important role, it would be undesirable to tamper with them.

Our proposed solution is to reduce the number of sheep in the beginning of the game. To test how many sheep is appropriate to remove 30 matches were played between two copies of the best agent from the previous experiments with 10000 simulations each(equivalent to about three seconds), but with different numbers of sheep at the start of the game. The heuristic function (3) was however altered to subtract the current amount of sheep and starting PST-value. So the factor $(S-20)$ in H_{mat} was altered to $(S-S_{start})$, and the factor $(V-38)$ in H_{pst} was altered to $(V-V_{start})$. One

sheep was removed at a time from the row furthest from the pasture until the win rate started to favor the foxes. The sheep were removed in the pattern shown in figure 9.



(a) One sheep removed (b) Two sheep removed (c) Three sheep removed

Figure 9: Starting positions in the Imbalance of Fox Game experiment

The win rate of the sheep decreases rapidly as sheep are removed from the game at the start. This shows how punishing it is for sheep to lose a piece without gaining terrain in return. But it also makes it hard to find an appropriate number of sheep to start with. As can be seen in figure 10, 18 starting sheep is the number of sheep that has a win rate closest to 50%. However, this number would only be suitable for skilled players. For novice players, 20 sheep is probably better because of the difficulty of playing as sheep.

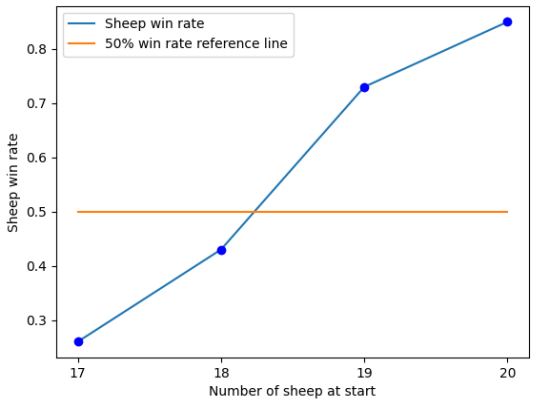


Figure 10: Sheep win rate with different number of sheep at start of game

6 DISCUSSION

Several discoveries were made from the experiments regarding both Fox Game and MCTS.

6.1 Strategies for Fox Game

As our initial knowledge of Fox Game and its different strategies was quite limited, a number of new and interesting discoveries were made during the work on this paper. Our first impression when playing the game against each other, as well as several testimonies online, suggested that the foxes are heavily favored. This is also what we found when the AI had a low number of simulations. But the results clearly show that when more advanced players, i.e. our AI with many simulations play, the sheep are as heavily favored. As we played more against each other, as well as against the AI, we also performed substantially better as sheep than as foxes.

One crucial tactic that made the AI considerably better than most humans, which we never managed to master, was to reliably capture one or both foxes. The AI does this by thinking many moves ahead, and by sacrificing some sheep in order to force the foxes into a position such that one of them has no available move, and is thereby captured.

One interesting outcome of this is that the foxes often try to counter this strategy by staying further back. This however might not be optimal when playing against human opponents that are not as skilled in the strategy of capturing the foxes. It could therefore be the case that the optimal agent for beating an AI opponent is not the same as the optimal agent for beating a human opponent that does not master the strategy of capturing foxes reliably.

Observations of the AI:s play suggest that it does not have an optimal strategy, but it has impeccable tactics. That is, it seems to play optimally in every given situation, but lacks a plan that lasts more than a couple of turns. It can for example not sense the long term consequences of letting a fox through its back line. This is in contrast to our strategy, that can beat the AI, in which we have a plan that lasts throughout the game.

This is most likely due to the positional part of the heuristic function being very simple and not valuing complex formations and how spread out or clumped together the sheep are. While we try to focus on the entire board, the AI seems to focus more locally around the area where the foxes, and thereby the action, is. This is something that a more complex heuristic function that uses a neural network could do better. The authors of [4] used the neural network approach with great success in the strongest Go-playing AI to date.

Although we have not mathematically solved Fox Game, our qualitative observations, as well as a limited number of games, suggest that it is solvable, and that the sheep can win every game. The work done in [2], where Fox-and-Geese is strongly solved, also suggests that the Scandinavian Fox Game could be solved since they are very similar games.

6.2 Optimal Agent for an Imbalanced Game

The heuristic function and cutoff experiments that compared different agents gave vague results that indicated that most of the alterations between agents did not affect the strength of the agents very much. This can partly be derived from the imbalance of the game. Throughout these experiments the sheep won 85% of the games. This meant that a strong agent playing as foxes would still have a very low win rate against a slightly weaker agent playing as sheep. It also meant that a strong agent playing as sheep does not have the same possibility to further increase its win rate, as the win rate of the sheep is naturally very close to 100%.

It would therefore be easier to find differences between agents playing a more balanced version of the game, for example the one proposed in section 5.5. It is however not certain that the same

constant-values of the agents are optimal for the balanced and imbalanced game. So one can not find an optimal agent for the balanced game, and claim that that agent is optimal for the original game as well.

An interesting aspect about Fox Game being asymmetrical is that it is not certain that the constant-values of the agent that is optimal for playing as sheep, are the same as the constant-values that are optimal for playing as foxes. It could for example be more beneficial for the foxes to search wider in the tree (which would require a larger constant C) compared to the sheep. It would therefore be interesting to examine if different constant-values of the agents are optimal for playing as sheep, compared to playing as foxes.

6.3 Optimal Cutoff

It is interesting to note the significant performance increase of the AI when a cutoff was introduced into the MCTS algorithm. Prior to that, the agents played terribly, and made huge mistakes which resulted in the foxes winning almost every game. Our reasoning for this difference in performance is that Fox Game is unusually unforgiving relative to similar games. The sheep dictates the pace of the game completely and the foxes cannot force the sheep to do anything. If the sheep keeps control of the game they are heavily favored, but it is often enough with one or two minor mistakes from the sheep for the foxes to take control of the game, and there are a lot of bad moves for the sheep to make. So it might be that the punishing nature of Fox Game also punishes the randomness in the MCTS playout where the sheep are bound to make big mistakes.

Another reason for the performance increase when introducing cutoff might be that the playouts lasts quite long (200-250 moves), and approximately the same game state arises several times during the same playout. This is because the sheep can move to the sides and then back again, and because if a few sheep in the back move to the side, the game state is functionally the same as before in many cases. This makes it hard to distinguish between a better and a worse starting position of the playout.

The experiments to find the optimal cutoff-value in section 5.3 yielded some noteworthy results. They show that while doing random playouts it is beneficial to use five as cutoff-value when both agents use the same number of simulations. Part two of the experiment however shows that the extra time spent doing the playout is not worth it, as a cutoff-value of zero outperforms a cutoff-value of five when given the same amount of time rather than the same number of simulations. This is a very interesting result as the creators of the world's foremost AI:s for playing Go, AlphaGo Zero, have come to the same conclusion for their AI [4].

6.4 Future Work

One interesting aspect to look further into is how to make a more sophisticated heuristic function. This can either be done manually by players who have more experience playing the game. This is however a cumbersome approach, and a better approach might be to train a neural network to evaluate the game state. As there are no online implementations of Fox Game, and therefore no recorded games, this would most likely have to be done via self play. It would be interesting to see how the effectiveness of self play is affected by the asymmetrical aspects of Fox Game. The resulting AI could for example end up in a situation where it plays good enough to always win as sheep, but still does not play optimally. And it would therefore be interesting to see if a neural network could be taught more quickly by playing the more even version of Fox Game suggested in section 5.5, and still perform well in the original version. An AI using a neural network could perhaps also find novel strategies that our AI, with its custom heuristic function, could not.

Another interesting future work would be to see if there are variations in the rules, other than those discussed in section 5.5, that would make the game more even. Such as altering the board, or the rules regarding how the pieces move. One could use the same AI proposed in this paper, or an improved version, to test the balance of these variations.

7 SUMMARY

As seen in the experiments above, a strong MCTS-agent for Fox Game requires a very low (or zero) cutoff-value, and therefore also a heuristic function. They also show that Fox Game is a very unbalanced game, where the sheep win a majority of games if both players play intelligently. This makes the development of a strong AI difficult because the difference in strength between two agents is hard to distinguish without huge amounts of data. It can also be seen that our best performing AI outplays humans with limited experience, and does well against more experienced players.

REFERENCES

[1] DeepMind. 2022. *AlphaGo.* https://deepmind.com/research/case-studies/alphago-the-story-so-far

[2] Stefan Edelkamp and Hartmut Messerschmidt. 2010. Strongly Solving Fox-and-Geese on Multi-core CPU. In *KI 2010: Advances in Artificial Intelligence*, Rüdiger Dillmann, Jürgen Beyerer, Uwe D. Hanebeck, and Tanja Schultz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 291–298.

[3] Claude E. Shannon. 1950. XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41, 314 (1950), 256–275. https://doi.org/10.1080/14786445008521796 arXiv:https://doi.org/10.1080/14786445008521796

[4] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature (London)* 529, 7587 (2016), 484–489.

[5] SpelRegler. 2022. *Rävspelet Regler.* https://www.spelregler.org/ravspelet-regler/

[6] Mark H. M. Winands. 2017. Monte-Carlo Tree Search in Board Games. In *Handbook of Digital Games and Entertainment Technologies*, Paolo Ciancarini Ryohei Nakatsu, Matthias Rauterberg (Ed.). Springer Singapore, Singapore, 47–74.